Naval Research Laboratory

Washington, DC 20375-5320

NRL/MR/5329--16-9656

NRL Radar Division C++ Coding Standard

JAMES B. EVINS

Advanced Radar Systems Branch
Radar Division

December 5, 2016

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
05-12-2016	NRL Memorandum Report	

4. TITLE AND SUBTITLE

NRL Radar Division C++ Coding Standard

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
63271N

6. AUTHOR(S)

James B. Evins

5d. PROJECT NUMBER
2913

5e. TASK NUMBER

5f. WORK UNIT NUMBER
53-9741-B-5

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
4555 Overlook Avenue, SW
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/MR/5329--16-9656

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research
875 N. Randolph Street, Suite 1425
Arlington, VA 22203

10. SPONSOR / MONITOR'S ACRONYM(S)

ONR

11. SPONSOR / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

A C++ coding standard is defined that should be adhered to when writing C++ code. It is primarily addressed to all those involved in the production of C++ code for Naval Research Laboratory (NRL) Radar Division software projects such as the Office of Naval Research (ONR) Integrated Topside (InTop) Innovative Naval Prototype (INP) Flexible Distributed Array Radar (FlexDAR) backend software. It was originally derived from the CERN C++ Coding Standard Specification, Version 1.1, dated 5 January 2000, but has been significantly modified and updated. It also draws on other sources and includes many local modifications for NRL Radar Division applications. The coding standard provides tools aimed at helping C++ programmers develop programs that are free of common types of errors, maintainable by different programmers, portable to other operating systems, easy to read and understand, and have a consistent style. Questions of design, such as how to design a class or a class hierarchy, are beyond the scope of this standard. It assumes the code is to be handwritten and not generated; otherwise, a different coding standard would be needed for the input to the code generator. This coding standard is not intended in any way as a substitute for the study of a book on C++ programming.

15. SUBJECT TERMS

C++ coding
ONR Integrated Topside (InTop) Program
NRL Radar Division
Flexible Distributed Array Radar (FlexDAR)
CERN C++ Coding Standard Specification

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			James B. Evins
Unclassified Unlimited	Unclassified Unlimited	Unclassified Unlimited	Unclassified Unlimited	51	**19b. TELEPHONE NUMBER** *(include area code)* (202) 404-1942

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Contents

1 Introduction

The approach adopted for the definition and development of this NRL Radar Division C++ Coding Standard is described here..

1.1 Purpose

The purpose of this document is to define a C++ coding standard that should be adhered to when writing C++ code for NRL Radar Division applications. The ISO 9000 and the Capability Maturity Model (CMM) state that coding standards are mandatory for any organization with quality goals. The purpose of this standard is to provide tools aimed at helping C++ programmers develop programs that are:

- Free of common types of errors
- Maintainable by different programmers
- Portable to other operating systems
- Easy to read and understand
- Have a consistent style

Questions of design, such as how to design a class or a class hierarchy, are beyond the scope of this document. It is assumed here that the code is to be handwritten and not generated; otherwise a different coding standard would be needed for the input to the code generator.

This document is not intended in any way as a substitute for the study of a book on C++ programming.

1.2 Intended Audience

This document is addressed to all those involved in the production of C++ code for NRL Radar Division software projects such as the FlexDAR backend software.

1.3 References

This document is derived from the following sources:

CERN	C++ Coding Standard Specification, Version 1.1, ID: CERN-UCO/1999/207, 5 January 2000
TMH	C++ Coding Standard, 2008-03-01, tmh@possibility.com: http://www.possibility.com/Cpp/CppCodingStandard.html
W[IS]	Wikipedia article on Indent Style: http://en.wikipedia.org/wiki/Indent_style
NRL	Local wisdom and “folklore.”

Manuscript approved September 29, 2016.

2 Naming

This section contains a set of conventions on how to choose, write, and administer the names for all entities over which the programmer has control. This should guarantee that programs are easier to understand, read, and maintain.

2.1 Naming of files

NF1 File Name Guidance

A filename should be the same as the module (usually a single class) that it defines. A single module can be defined by as many as three files using the following suffixes:

".h" – header or specification file
".cpp" – implementation file
".inl" – implementation file for inline methods

When using an ".inl" file, it should be included from the corresponding ".h" file after the class definition.

Example:

The class "CalorimeterCluster" would be defined by the following files:

```
CalorimeterCluster.h
CalorimeterCluster.cpp
CalorimeterCluster.inl (included from CalorimeterCluster.h)
```

Source CERN, NRL (resolved suffix choices and condensed three rules into one).

2.2 Meaningful Names

NM1 General Naming Guidance

Names of classes, methods, and important variables should be chosen with care, and should be meaningful. Abbreviations are to be avoided, except where they are widely accepted.

These are critical in making the code easy to read and use.

Source CERN.

NM2 Class Name Guidance

- Names should be short but descriptive.
- Name the class after what it is. If you can't think of what it is, that is a clue that you have not adequately thought through the design.
- Compound names of more than three words are a clue that your design may be confusing various entities in your system. Revisit your design.
- Avoid the temptation of bringing the name of the class, from which a class is derived, into the derived class's name. A class should stand on its own. It doesn't matter what it is derived from.
- Suffixes are sometimes helpful. For example, if your system uses agents then naming something "DownloadAgent" conveys real information.

Source TMH.

NM3 Method and Function Name Guidance

Usually every method and function performs an action, so the name should make clear what it does: "checkForErrors()" instead of "errorCheck()", "dumpDataToFile()" instead of "dataFile()." This will also make functions and data objects more distinguishable.

Classes are often nouns. By making function names verbs and following other naming conventions, programs can be read more naturally.

Suffixes are sometimes useful, e.g.:

Max – to mean the maximum value something can have.
Key – key value.

Prefixes are also sometimes useful, e.g.:

Is – to ask a question about something.
Get – to get a value.
Set – to set a value.

Source TMH.

NM4 Variable Name Guidance

Variable names should be short but descriptive. For contexts that are general in nature, short algebraic names such as “x,” “y,” or “i” may be appropriate. However, in more specific contexts names should be more descriptive.

Example 1:

```
// Example of a general context
double sumVector( double a[], int n )
{
    double sum = 0;
    for ( int i = 0; i < n; i++ )
    {
        sum += a[i];
    }
    return sum;
}

// A contrived example of a similar but specific context
double calculateMeanScore( double scores[], int nScores )
{
    double scoreSum = 0;
    for ( int iScore = 0; iScore < nScores; iScore++ )
    {
        scoreSum += score[iScore];
    }
    return scoreSum / nScores;
}
```

For indexes and limits it is useful to use “i” and “n” as base names or prefixes, respectively. For example, “iElement” and ”nElements.”

Example 2:

```
const int nElements = 10;

Element a[nElements];

for ( int iElement = 0; iElement < nElements; iElement++ )
{
    // a[iElement] = …
}
```

Source NRL.

2.3 Illegal Naming

NI1 **Do not create very similar names.**

Very similar variable names or type names might cause confusion in reading the code. In particular, do not create names that differ only by case.

Example:

track, Track, TRACK cmlower, cslower

Source CERN.

NI2 **Do not use identifiers that begin with an underscore.**

Many identifiers of this kind are reserved C keywords.

Source CERN.

NI3 **Avoid single and simple character names (e.g. "j" "iii").**

Possible exceptions:

- Local loop and array indices.
- Inside small generalized functions that are more easily expressed in algebraic terms. See NM4.

Source CERN, NRL.

2.4 Naming Conventions

NC1 **Use namespaces to avoid name conflicts.**

A name conflict occurs when a name is defined in more than one place. For example, two different class libraries could give two different classes the same name. If you try to use many class libraries at the same time, there is a fair chance that you will be unable to compile and link the program because of name conflicts. You can avoid that by declaring and defining names (that would otherwise be global) inside namespaces.

Example:

A namespace is a declarative region in which classes, functions, types, and templates can be defined.

```
namespace emc {
class Track { … };
    // …
}
```

A name qualified with a namespace name refers to a member of the namespace.

```
emc::Track electronTrack;
```

A "using" declaration makes it possible to use a name from a namespace without the scope operator.

```
using emc::Track;    // using declaration
Track electronTrack;
```

It is also possible to make all names from a namespace accessible with a "using" directive.

```
using namespace emc;    // using directive
Track electronTrack;    // emc::Track electronTrack;
Array<Track> allTracks; // emc::Array<emc::Track> allTracks;
```

Source CERN.

The following items might appear rather arbitrary but the importance of these conventions is simply in fostering a common style and naming across a wide community of programmers. The benefit is an increase in the readability and maintainability of the produced code, especially when compared to a situation where each programmer adopts their own naming convention.

NC2 **Start class names, typedefs, and enum types with an uppercase letter.**

Example:

```
class Track;
typedef vector<MCParticleKinematics*> TrackVector;
enum State { green, yellow, red };
```

Source CERN.

NC3 **Start names of variables, functions, and namespaces with a lowercase letter.**

Example:

```
double energy;
void extrapolate();
```

Source CERN.

NC4 **In names that consist of more than one word, write the words together, and start each word that follows the first one with an uppercase letter.**

Example:

```
class OuterTrackerDigit;
double depositedEnergy;
void findTrack();
```

Source CERN.

NC5 **Include units in names.**

If a variable represents time, weight, or some other unit, include the unit in the name so developers can more easily spot problems.

Example:

```
uint32 mTimeoutMsecs;
uint32 mMyWeightLbs;
```

Source TMH.

NC6 No all uppercase abbreviations.

When confronted with a situation where you could use an all uppercase abbreviation instead use an initial uppercase letter followed by all lowercase letters. No matter what.

Justification: People seem to have very different intuitions when making up names containing abbreviations. It's best to settle on one strategy so the names are absolutely predictable.

Take, for example, "NetworkABCKey." Notice how the "C" from "ABC" and the "K" from "key" are confused. Some people don't mind this but others hate it so you'll find different policies in different code and you never know what to call something.

Example:

```
class FluidOz              // NOT FluidOZ
class NetworkAbcKey        // NOT NetworkABCKey
```

Source TMH.

NC7 Use simple prefixes to indicate scope and some other common characteristics of variables.

- Use 'm' for member variables.
- Use 'p' for pointers.

Example:

```
class Example
{
    public:
        Example();

        void setOffset( int offset );
        void setWidget( Widget* pWidget );

    private:
        string     mLabel;
        int        mOffset;
}
```

Source TMH, NRL.

3 Coding

This section contains a set of items regarding the "content" of the code. Organization of the code, control flow, object life cycle, conversions, object-oriented programming, error handling, parts of C++ to avoid, and portability, are all examples of issues that are covered here.

The purpose of the following items is to highlight some useful ways to exploit the features of the programming language, and to identify some common or potential errors to avoid.

3.1 Organizing the Code

CO1 **Each header file should be self-contained.**

If a header file is self-contained, nothing more than the inclusion of the single header file is needed to use the full interface of the class defined.

To enforce this rule, the first thing to include in an implementation file should be its corresponding header file. All modules should have an implementation section, even if just to include its corresponding header file.

Example:

The file Track.cpp, it would include Track.h before any other header files:

```
#include "Track.h"  // First include

#include <cmath>    // All other includes
#include <sstream>
#include <iomanip>
#include "Other.h"
…
```

Source CERN.

CO3 **Header files should begin and end with multiple-inclusion protection.**

This is implemented as follows:

```
#ifndef IDENTIFIER_h
#define IDENTIFIER_h

// The text of the header goes in here …

#endif // IDENTIFIER_h
```

The actual value for the IDENTIFIER is typically the concatenation of the namespace and class defined in the header file, separated by an underscore (_).

Header files are often included many times in a program. Because C++ does not allow multiple definitions of a class, it is necessary to prevent the compiler from reading the definitions more than once.

Example:

The file Track.h in the emc subsystem might look like this:

```
#ifndef emc_Track_h
#define emc_Track_h

namespace emc
{
    class Track
    {
        // Methods and Member Definitions of emc::Track …
    }
}

#endif // emc_Track_h
```

Source CERN, NRL.

CO4 **Each header file should contain one class (or possibly a set of small tightly coupled classes) declaration(s) only. The class may also include small embedded class definitions.**

This makes it easier to read your source-code files. This also improves the version control of the files; for example the file containing a stable class declaration can be committed and not changed anymore.

Source CERN.

CO5 **Implementation files should hold the member function definitions for a single class (or possibly a set of small tightly coupled classes) as defined in the corresponding header file.**

This is for the same reason as for item CO4.

Source CERN.

3.2 Control Flow

CF1 **Do not change a loop variable inside a for-loop block.**

When you write a for-loop, it is highly confusing and error prone to change the loop variable within the loop body rather than inside the expression executed after each iteration.

Source CERN.

CF2 **Follow all flow control primitives (if, else, while, for, do, switch, and case) by a block, even if it is empty.**

This makes code much more reliable and easier to read.

Example:

```
while (condition)
{
    statement;
}
```

Avoid the following error-prone form:

```
if (condition) // avoid! this omits the braces {}!
    statement;
```

Source CERN.

CF3 **All switch statements should have a default clause.**

In some cases, the default clause can never be reached because there are case labels for all possible enum values in the switch statement. However, by having such an unreachable default clause, you show a potential reader that you know what you are doing and you also provide for future changes. If an additional enum value is added, the switch statement should not just silently ignore the new value. Instead, it should in some way notify the programmer that the switch statement must be changed; for example, you could throw an exception.

Example:

```
// somewhere specified: enum Colors { GREEN, RED }

// semaphore of type Colors

switch(semaphore)
{
    case GREEN:
        // statement
        // break;
    case RED:
        // statement
        // break;
    default:
        // unforseen color; it is a bug
        // do some action to signal it
}
```

Source CERN.

CF4 **All compound "if" statements should have a final "else" clause.**

This makes code much more readable and reliable by clearly showing the flow paths. The addition of a final "else" is particularly important in the case where you have "if/else-if."

Example:

```
if (val == ThresholdMin)
{
    statement;
}
else if (val == ThresholdMax)
{
    statement;
}
else
{
    statement; // handles all other(unforseen)cases
}
```

Source CERN.

CF5 Do not use "goto."

Use break or continue instead.

This statement also remains valid in the case of nested loops, where the use of control variables can easily allow for breaking the loop, without using "goto."

Source CERN.

CF6 Do not have overly complex functions.

The number of possible paths through a function, which depends on the number of control flow primitives, is the main source of function complexity. Therefore, be aware that heavy use of control flow primitives will make your code more difficult to maintain.

Most functions should be very short. As a rule of thumb, they should typically not be longer than 10 lines. When not practical to keep functions this short, the absolute upper limit should be about 50 lines (should be contained on a single printed page). Additionally, to avoid text-wrapping, line lengths should be limited to no more than 120 columns.

Source CERN, NRL.

3.3 Object Life Cycle

In this section, it is suggested how objects are best declared, created, initialized, copied, assigned, and destroyed.

3.3.1 Initialization of Variables and Constants

CL1 Declare variables initialized to numeric values or strings in a highly visible position; whenever possible collect them in one place.

It would be very hard to maintain a code in which numeric values or strings are spread over a large file. If the declaration and initialization of variables to numeric values or strings are placed in the most visible position, it will be easy to locate and maintain them.

Source CERN.

CL2 Declare each variable with the smallest possible scope and initialize it at the same time.

It is best to declare variables close to where they are used. Otherwise, you may have trouble finding out the type of a particular variable.

It also very important to initialize the variable immediately, so that its value is well defined.

Example:

```
int value = -1; // initial value clearly defined

int maxValue;   // initial value undefined
                // NOT recommended
```

Source CERN.

CL3 **In the function implementations, do not use numeric values or string literals; use symbolic values instead.**

Exceptions to this rule would be simple constants such as true, false, "", -1, 0, 1 or 2 if their meaning is clear and they don't represent something that could possibly be tweaked in a different application.

Do not use symbolic values such as "two" to represent the value 2. The symbolic value should be descriptive of what the value represents such as "nSamples."

Example:

```
const int nValues = 2;

double values[nValues];

for ( int iValue = 0; iValue < nValues; iValue++ )
{
    // Do something with values[iValue] …
}
```

instead of

```
double values[2];

for ( iValue = 0; iValue < 2; iValue++ )
{
    // Do something with values[iValue] …
}
```

however, the use of the numeric value of 2 in the following example would be ok:

```
const double pi = 3.1415926;

wRads = 2 * pi * fHz;
```

Source CERN, NRL.

CL4 **Do not use the same variable name in both outer and inner scope.**

Otherwise, the code would be very hard to understand; and it would certainly be a major error-prone condition.

Source CERN.

CL5 **Declare each variable in a separate declaration statement.**

Declaring multiple variables on the same line is not recommended. The resulting code will be difficult to read and understand.

Some common mistakes should also be avoided. Remember that when you declare a pointer, a unary pointer is bound only to the variable that immediately follows.

Example:

```
int i, *ip, ia[100], (*ifp)(); // Not recommended

// recommended way:

LoadModule* oldLm = 0; // pointer to the old object
LoadModule* newLm = 0; // pointer to the new object
```

Source CERN.

3.3.2 Constructor Initializer Lists

CL6 **Initialize all data members in the class constructors.**

If you add a new data member, don't forget to update all constructors, operators, and the destructor.

Source CERN.

3.3.3 Copying of Objects

CL8 **Avoid unnecessary copying of objects that are costly to copy.**

Because a class could have other objects as data members or inherit from other classes, many member function calls would be needed to copy the object. To improve performance, you should not copy an object unless it is necessary.

It is possible to avoid copying by using pointers and references to objects, but then you will instead have to worry about the lifetime of objects. You must understand when it is necessary to copy an object and when it is not.

Source CERN.

CL9 **A function must never return, or in any other way give access to, references or pointers to local variables outside the scope in which they are declared.**

Returning a pointer or reference to a local variable is always wrong because it gives the user a pointer or reference to an object that no longer exists.

Source CERN.

CL10 **If objects of a class should never be copied, then the copy constructor and the copy assignment operator should be declared private and not implemented.**

Ideally the question of whether the class has a reasonable copy semantic will naturally come out of the design process. Do not push copy semantics on a class that should not have it.

By declaring the copy constructor and copy assignment operator as private, you can make a class non-copyable. They do not have to be implemented, only declared.

Source CERN.

CL11 **If objects of a class should be copied, then the copy constructor and the copy assignment operator should be implemented with the desired behavior.**

The compiler will generate a copy constructor, a copy assignment operator, and a destructor if these member functions have not been declared. A compiler-generated copy constructor does member-wise initialization and a compiler-generated copy assignment operator does member-wise assignment of data members and base classes. For classes that manage resources (examples: memory (new), files, sockets) the generated member functions have probably the wrong behavior and must be implemented. You have to decide if the resources pointed to must be copied as well (deep copy), and write the right behavior in the operators.

Of course, constructor and destructor must be implemented as well, see item CB2.

Source CERN.

CL12 **Assignment-member functions should work correctly when the left and right operands are the same object.**

This requires some care when writing assignment code, as the case when left and right operands are the same may require that most of the code is bypassed.

Example:

```
A& A::operator=(const A& a) {

    if (this != &a) // beware of s=s
    {

        // … implementation of operator=
    }
}
```

Source CERN.

3.4 Conversions

CC1 **Use explicit rather than implicit type conversion.**

Most conversions are bad in some way. They can make the code less portable, less robust, and less readable. It is therefore important to use only explicit conversions. Implicit conversions are almost always bad.

Source CERN.

CC2 **When the new casts are supported by the compiler, use the new cast operators (dynamic_cast and static_cast) instead of the C-style casts.**

The new cast operators give the user a way to distinguish between different types of casts. Their behavior is well defined in situations where the behavior of an ordinary cast is undefined, or at least ambiguous.

Source CERN.

CC3 **Do not convert const objects to non-const.**

In general you should never cast away the constness of objects.

The only rare case when you have to do it is in the case where you need to invoke a function that has incorrectly specified a parameter as non-const even if it does not modify it. If the correction of this function is really impossible, then use the cast operator “const_cast.”

Source CERN.

3.5 The Class Interface

The class interface is the most important part of the class. Sophisticated algorithms will not help if the class interface is wrong.

3.5.1 Inline Functions

CI1 Inline access functions and forwarding functions.

Inline functions can improve the performance of your program. However, they can increase the overall size of the program and then, in some cases, have the opposite result. It can be hard to know exactly when inlining is appropriate. In general, inline only very simple functions, such as accessor methods.

Source CERN.

3.5.2 Argument Passing and Return Values

CI2 Adopt the good practice of design functions without any side effects.

Example:

No-one would expect `sin(x)` *to modify* `x`.

Source CERN.

CI3 Pass arguments of built-in types by value unless the function should modify them.

A good practice is to pass built-in types such as char, int, and double by value because it is cheap to copy such variables. This recommendation is also valid for some objects of classes that are cheap to copy, such as simple aggregates of very small number of built-in types.

Example:

```
void func(char c);           // OK
void func(int i);            // OK
void func(double d);         // OK
void func(complex<float> c); // OK
```

Source CERN.

CI4 **Pass arguments of class types by reference or pointer.**

Arguments of class type are often costly to copy so it is convenient to pass a reference (or in some cases a pointer), preferably declared const, to such objects; in this way the argument is not copied. Const access guarantees that the function will not change the argument.

Example:

```
void func(const LongString& s); // const reference
```

Source CERN.

CI5 **Have the operator "=" return a reference to `*this`.**

This ensures that:

```
a = b = c;
```

will assign c to b and then b to a as is the case with built in objects.

Source CERN.

3.5.3 Const Correctness

CI6 **Declare a pointer or reference argument, passed to a function, as const if the function does not change the object bound to it.**

An advantage of const-declared parameters is that the compiler will actually give you an error if you modify such a parameter by mistake, thus helping you to avoid bugs in the implementation.

Example:

```
// operator<< does not modify the String parameter
ostream& operator<<(ostream& out, const String& s);
```

Source CERN.

CI7 **The argument to a copy constructor and to an assignment operator should be a `const` reference.**

This ensures that the object being copied is not altered by the copy or assign.

Source CERN.

CI8 **In a class method, do not return a pointer or non-`const` reference to private data members.**

Otherwise you break the principle of encapsulation.

If necessary, you can return the pointer to `const` or `const` reference.

Source CERN.

CI9 **Declare as post `const` a member function that does not affect the state of the object.**

Declaring a member function as `const` has two important implications:

- Only const member function can be called for `const` objects.
- A `const` member function will not change data members.
-

It is a common error to forget to `const` declare member functions that should be `const`.

Source CERN.

CI10 **Do not let `const` member functions change the state of the program.**

A `const` member function promises not to change any of the data members of the object. Usually this is not enough. It should be possible to call a `const` member function any number of times without affecting the state of the complete program. It is therefore important that a `const` member function refrain from changing static data members, global data, or other objects to which the object has a pointer or reference.

Source CERN.

3.5.4 Overloading and Default Arguments

CL11 **Use function overloading only when methods differ in their argument list but the task performed is the same.**

Using function-name overloading for any other purpose than to group closely related member functions is very confusing and is not recommended.

Source CERN.

3.6 Memory Management

CN1 **Match every invocation of `new` with one invocation of `delete` in all possible control flows from `new`.**

A missing `delete` would cause a memory leak.

Example:

If you allocate memory in the constructor, you should take care of deallocate it in the destructor.

Source CERN.

CN2 **A function must not use the `delete` operator on any pointer passed to it as an argument.**

NOTE: This is also to avoid dangling pointers, i.e. pointers to memory which has been given back. Such code will often continue to work until the memory is re-allocated for another object.

Source CERN.

CN3 **Do not access a pointer or reference to a deleted object.**

A pointer that has been used as argument to a `delete` expression should not be used again unless you have given it a new value, because the language does not define what should happen if you access a deleted object. You could assign the pointer to 0 or a new valid object. Otherwise, you get a "dangling" pointer.

Source CERN.

3.7 Static and Global Objects

CS1 **Do not declare global variables.**

If necessary, encapsulate those variables in a class or in a namespace.

Source CERN.

CS2 **Use global functions only for symmetric binary operators.**

This is the only way to get conversions of the left operand of binary operations to work. It is common in implementing the symmetric operator to call the corresponding asymmetric binary operator.

Example:

```
Complex operator* (const Complex & lhs, const Complex & rhs)
{
    Complex result(lhs);
    return result *= rhs;
}
```

*Here the * operator has been defined for Complex numbers in terms of the *= operator.*

Source CERN

3.8 Object-Oriented Programming

CB1 **Declare data members private or protected.**

This ensures that data members are only accessed from within member functions. Hiding data makes it easier to change implementation and provides a uniform interface to the object.

Example:

```
class Point
{
    public:
        Number x() const; // Return the x coordinate

    private:
        Number mX;       // The x coordinate (safely hidden)
};
```

The fact that the class Point has a data member mX which holds the x coordinate is hidden.

Source CERN.

CB2 **Always declare and implement constructor and destructor.**

This is important to avoid possible memory leak problems that one would not expect.

Example:

```
class Track
{
    public:
        Track();
        virtual ~Track();

    private:
        class1 c1;
        class2 c2;
        int i3;
};

Track::Track() : c1(), c2(), i3(0)
{
    // Empty
}

Track::~Track()
{
    // Empty
}
```

Source CERN.

CB3 **A public base class must have either a public virtual destructor or a protected destructor.**

The destructor is a member function that in most cases should be declared virtual. It is necessary to declare it virtual in a base class if derived class objects are deleted through a base class pointer. If the destructor is not declared virtual, only the base class destructor will be called when an object is deleted that way.

However, there is a case where it is not appropriate to use virtual destructor: mix-in classes. Such a class is used to define a small part of an interface, which is inherited (mixed in) by subclasses. In these cases, the destructor, and hence the possibility of a user deleting a pointer to such a mix-in base class, should normally not be part of the interface offered by the base class. It is best in these cases to have a non-virtual, non-public destructor because that will prevent a user of a pointer to such a base class from claiming ownership of the object and deciding to simply delete it. In such cases, it is appropriate to make the destructor protected. This will stop users from accidentally deleting an object through a pointer to the mix-in base-class, so it is no longer necessary to require the destructor to be virtual.

Source CERN.

CB4 **Always redeclare virtual functions as virtual in derived classes.**

This is just for clarity of code. The compiler will know it is virtual, but the human reader may not. This, of course, also includes the destructor, as stated in item CB3.

Source CERN.

CB5 **Avoid multiple inheritance.**

Multiple inheritance is seldom necessary, and it is rather complex and error prone.

The only valid exception is for inheriting interfaces or when the inherited behavior is completely decoupled from the class's responsibility.

Source CERN.

CB6 **Use public inheritance.**

Private and protected inheritance is useful in rather specific cases only. As a rule of thumb use aggregation instead.

Source CERN.

CB7 **Avoid the use of friend declarations.**

Friend declarations are symptoms of bad design and they break encapsulation. Typically, you can solve your problem in a different way. An exception to this rule is when defining operators outside of the class.

Source CERN.

3.9 Assertions and Error Conditions

CE1 **Preconditions and postconditions should be checked for validity.**

You should validate your input and output data, whenever an invalid input can cause an invalid output.

Example:

```
EmcString::EmcString(const char* cp) throw(bad_alloc)
    : lengthM(strlen(cp))
{
    // check of preconditions: cp != 0
    // …

    // check of postconditions:
    // operator new() will throw bad_alloc
    // if allocation fails

    cpM = new char[lenthM + 1];
    strcpy(cpM, cp);
}
```

Source CERN.

CE2 **Remove all assertions from production code.**

Assertions should only be used for the testing phase. The program will also run faster if unnecessary checks are removed.

Some conditions are not checked by assertions. You should not use assertions to check conditions that should always result in throwing an exception if the check fails. Such exceptions are part of the production code and should not be removable.

Source CERN.

3.10 Error Handling

CH1 **Check for all errors reported from functions.**

It is important to always check error conditions, regardless of how they are reported. If a function throws exceptions, it is important to catch all of them.

Source CERN.

CH2 **Use exception handling instead of status values and error codes.**

For error reporting, exception handling is a more powerful technique than returning status values and error codes. It allows the separation of code that handles errors from the ordinary flow of control.

Because an exception is an object, an arbitrary amount of error information can be stored in an exception object; the more information that is available, the greater the chance that the correct decision will be made for how to handle the error.

In certain cases, exception handling can be localized to one function along a call chain; this implies that less code needs to be written, and it is more legible.

Example:

```
try{
    // ordinary flow of control
    f();
    g();
}
catch(…)
{
    // handler for any kind of exception
    // error handling
}
```

Source CERN.

CH3 **Do not throw exceptions as a way of reporting uncommon values from a function.**

Your code can be difficult to understand if you throw exceptions in many different situations, ranging from a way to report unusual threads in your code to reporting fatal runtime problems.

Example:

Take the case of a function `find()`*. It is quite common that the object looked for is not found, and it is certainly not a failure; it is therefore not reasonable in this case to throw an exception. It is clearer if you return a well-defined value.*

Source CERN.

CH4 **Use exception specifications to declare which exceptions might be thrown from a function.**

If a function does not have an exception specification, that function is allowed to throw any type of exception; that makes code unreliable and difficult to understand.

It is recommended that exception specification be used as much as possible. The compiler will check that the exception classes exist and are available to the user. Compilers are also sometimes able to detect inconsistent exception specification during compilation.

Example:

```
char& EmcString::at(size_t index) throw(EmcIndexOutOfRange)
{
    if (index > lengthM)
    {
        throw EmcIndexOutOfRange(index);
    }
    return cpM[index];
}
```

Source CERN.

3.11 Parts of C++ to Avoid

This section highlights parts of the language that should be avoided, because there are better ways to achieve the desired results.

CA1 **Use `new` and `delete` instead of `malloc`, `calloc`, `realloc`, and `free`.**

You should avoid all memory-handling functions from the standard C-library (`malloc`, `calloc`, `realloc`, and `free`) because they do not call constructors for new objects or destructors for deleted objects.

Source CERN.

CA2 **Use the `iostream` functions rather than those defined in `stdio`.**

`scanf` and `printf` are not type-safe and they are not extensible. Use `operator>>` and `operator<<` instead.

Source CERN.

CA3 **Do not use the ellipsis notation.**

Functions with an unspecified number of arguments should be avoided because they are a common cause of bugs that are hard to find.

Example:

```
// avoid to define functions like:

void error(int severity ...) // “severity” followed by a
                             // zero-terminated list of char*s
```

Source CERN.

CA4 **Do not use preprocessor macros, except for system-provided macros.**

Use templates or inline functions rather than the preprocessor macros.

Example:

```
// NOT recommended to have function-like macro
#define SQUARE(x) x*x

Better to define an inline function:

inline int square(int x)
{
    return x*x;
};
```

Source CERN.

CA5 **Do not use `#define` to define symbolic constants or `enum`s.**

Example:

```
#define levels 5 // NOT recommended

If you need to define a symbolic constant, use:

const int levels = 5;
```

Source CERN.

CA6 Use enum for related constants rather than const.

The enum construct allows a new type to be defined and hides the numerical values of the enumeration constants.

Example:

```
enum State {halted, starting, running, paused};
```

Source CERN.

CA7 Use the integer constant "0" for the null pointer; don't use `NULL`.

No object is allocated with the address "0." Consequently, "0" acts as a pointer literal, indicating that a pointer doesn't refer to an object. In C, it has been popular to define a macro `NULL` to represent the "0" pointer. Because of C++'s tighter type checking, the use of plain "0," rather than any suggested `NULL` macro, leads to fewer problems.

Source CERN.

CA8 Use the standard library (STL) whenever it has the desired functionality.

In particular, do not use `const char*` or built-in arrays "`[]`".

Source CERN.

CA9 Do not use `union` types.

Unions can be an indication of a non-object-oriented design that is hard to extend. The usual alternative to unions is inheritance and dynamic binding. The advantage of having a derived class representing each type of value stored is that the set of derived class can be extended without rewriting any code. Because code with unions is only slightly more efficient, but much more difficult to maintain, you should avoid it.

Source CERN.

CA10 Do not use asm (the assembler macro facility of C++).

Source CERN.

CA11 **Do not use the keyword `struct`.**

A `class` is identical to a `struct` except that by default its contents are private rather than public.

Source CERN.

CA12 **Do not use file scope objects; use class scope instead.**

File scope is a useless complication that is better to avoid.

Source CERN.

CA13 **Use the `bool` type of C++ for Booleans.**

Programmers may tend to use `int` instead of `bool` as this is a relatively new feature.

Source CERN.

CA14 **Avoid pointer arithmetic.**

Pointer arithmetic makes readability very difficult and it is certainly one of the most error prone parts.

Source CERN.

3.12 Readability and maintainability

CR1 **Avoid duplicated code and data.**

This statement has a twofold meaning.

The first, and most evident, is that one must avoid simply cutting and pasting pieces of code. When similar functionalities are necessary in different places, those should be collected in class methods and reused.

The second meaning is at the design level, and is the concept of code reuse.

Reuse of code has the benefit of making a program easier to understand and to maintain. An additional benefit is better quality because code that is reused is tested much better.

Source CERN.

CR2 **Optimize code only when you know you have a performance problem.**

This means that during the implementation phase you should write code that is easy to read, understand, and maintain. Do not write cryptic code just to improve its performance.

Performance problems are more likely solved at an architecture and design level.

Source CERN.

3.13 Portability

CP1 **All code must be adherent to the ANSI C++ standard.**

NOTE: Adhesion to the standard must be done to the extent that the selected compilers allow it. For gcc, use of –Wall –Werror and –pedantic options can help identify non-standard use.

Source CERN.

CP2 Make nonportable code easy to find and replace.

Isolate nonportable code as much as possible so that it is easy to find and replace. The use of "#ifdef" distributed throughout your code (a.k.a. ifdef-hell) should be avoided if at all possible. Platform specific code should be isolated to separate source files, such that ifdefs do not obscure the readability of the code.

Example 1:

A different set of header files must be included for windows and linux when implementing the Xyz *module. Instead of putting ifdefs around the includes, create header file called* 'Xyz_platform.h'. *This file will contain the following:*

```
#ifdef _WIN32
#include "Xyz_platform_windows.h"
#else
#include "Xyz_platform_linux.h"
#endif
```

This will localize the ifdef tree to 'Xyz_platform.h' and the platform specific includes to 'Xyz_platform_windows.h' and 'Xyz_platform_linux.h'.

Example 2:

Two methods of the Xyzzy *class require platform specific implementations for linux and vxWorks. All other methods can use the same implementation. The* Xyzzy *class would be implemented in three separate files:*

Xyzzy.cpp – implementation of all common methods.
Xyzzy_linux.cpp – linux implementation of non-common methods.
Xyzzy_vxworks.cpp – vxWorks implementation of non-common methods.

The build system should select which platform specific implementation to compile and link based on the current target architecture.

Source CERN and NRL.

CP3 **Headers supplied by the implementation (system or standard libraries header files) should go in <> brackets; all other headers should go in "" quotes.**

Example:

```
// Include only standard header with <>
#include <iostream>  // OK: standard header
#include <MyFyle.hh> // NO: nonstandard header

// Include any header with ""
#include "stdlib.h"  // NO: better to use <>
#include "MyFyle.hh" // OK
```

Source CERN.

CP4 **Do not specify absolute directory names in include directives.**

It is better to specify to the build environment where files may be located because then you do not need to change any include directives if you switch to a different platform.

Source CERN.

CP5 **Always treat include file names as case-sensitive.**

Some operating systems, e.g. Windows, do not have case-sensitive file names. You should always include a file as if it were case-sensitive. Otherwise your code could be difficult to port to an environment with case-sensitive file names.

Example:

```
// Includes the same file on Windows, but not on UNIX
#include <Iostream>
#include <iostream>
```

Source CERN.

CP6 **Do not make assumptions about the size or layout of an object in memory.**

The sizes of built-in types may be different in different environments. For example, an `int` may be 16, 32, or even 64 bits long. The memory layout of objects may also be different in different environments, so it is unwise to make any kind of assumptions about the layout of objects in memory, such as when concatenating data types of different sizes.

Source CERN.

CP7 **Do not cast a pointer to a shorter quantity to a pointer to a longer quantity.**

Certain types have alignment requirements, which are requirements about the address of objects. For example, some architectures require that objects of a certain size start at an even address. It is a fatal error if a pointer to an object of that size points to an odd address. For example, you might have a char pointer and want to convert to an int pointer. If the pointer points to an address that it is illegal for an int, dereferencing the int pointer creates a runtime error.

Source CERN.

CP8 **Take machine precision into account in your conditional statements. Have a look at the numeric_limits<T> class, and make sure your code is not platform dependent. In particular, take care when testing floating-point values for equality.**

Example:

It is better to use:

```
const double TOLERANCE = 0.001;
…
#include <cmath>
if ( fabs(value1 – value2) < TOLERANCE ) …
```

than

```
if ( value1 == value2 ) …
```

Source CERN.

CP9 **Do not depend on the order of evaluation of arguments to a function.**

The order of evaluation of function arguments is strongly compiler dependent.

In particular never use `++`, `--` operators on method arguments in function calls. The behavior of `foo(a++, vec(a))`; is platform dependent.

Example:

```
func(f1(), f2(), f3());
// f1 may be evaluated before f2 and f3,
// but don’t depend on it!
```

Source CERN.

CP10 **Avoid using system calls if there is another possibility (e.g. the C++ runtime library).**

For example, do not forget about non-Unix platforms.

4 Style

Code is always written in a particular style. Discussing style is highly controversial. This section contains indications aimed at defining one style; that should allow a common and consistent "style of the code,",i.e. a common look. Style relates to matters which do not affect the output of the compiler.

4.1 General aspects of style

SG1 **The `public`, `protected`, and `private` sections of a class should be declared in that order. Within each section, nested types (e.g. `enum` or `class`) should appear at the top.**

The public part should be most interesting to the user of the class, and should therefore come first. The private part should be of no interest to the user and should therefore be listed last in the class declaration.

Example:

```
class Path
{
    public:
        Path();
        ~Path();

    protected:
        void draw();

    private:
        class Internal
        {
            // Path::Internal declarations go here …
        };
};
```

Source CERN.

SG2 **Keep the ordering of methods in the header file and in the source files identical.**

This facilitates the readability of the class implementation.

Source CERN.

SG3 **Arrange long statements on multiple lines in a way which maximizes readability. If possible, break very long statements up into multiple ones.**

Source CERN.

SG4 **Do not have any method bodies inside the class definitions (in header files).**

The class definition will be more compact and comprehensible if no implementation can be seen in the class interface.

This also applies to inline functions. You can either put them in a separate file or at the end of the header file below the class definition.

Example:

```
class X
{
    public:
        // Not recommended: function definition in class
        bool insideClass() const { return false; }
        bool outsideClass() const;
};

// Recommended: function definition outside class
inline bool X::outsideClass() const
{
    return true;
}
```

Source CERN.

SG5 **Include meaningful dummy argument names in function declarations.**

Although they are not compulsory, dummy arguments greatly improve the understanding and use of the class interface.

Example:

The constructor below takes 2 Numbers, but what are they?

```
class Point
{
    public:
        Point( Number, Number );
}
```

the following is clearer

```
class Point
{
    public:
        Point( Number x, Number y );
}
```

because it is explicitly indicated the meaning of the parameters.

Source CERN.

SG6 **Any dummy argument names used in function declarations should be the same as in the implementation.**

Source CERN.

SG7 **The code must be properly indented in a consistent style for readability purposes.**

It is recommended that the "Allman" (a.k.a. "BSD") style be used in all C/C++ code. This style puts the brace associated with a control statement on the next line, indented to the same level as the control statement. Statements within the braces are indented to the next level. See http://en.wikipedia.org/wiki/Indent_style#Allman_style for more information.

It is recommended that each level of indentation be 8 spaces. Tab characters should never be used for indentation.

Example:

```
for ( int i=1; i<n; i++ )
{
        a[i] = b[i] + 200;
}

if ( c.isValid() )
{
        if ( d.isEmpty() && c.value() == 0 )
        {
                f( c, d );
        }
}
```

There are a number of code beautifier tools that can be used to help maintain these and other style elements. One such tool is the "Artistic Style" or astyle GNU/Linux utility. The following ~/.astylerc file can be used to help ensure that the indentation style is consistent with this guideline.

```
-A1
--indent=spaces=8
--indent-classes
--indent-namespaces
--indent-col1-comments
--align-pointer=type
--align-reference=type
--lineend=linux
```

Source CERN, NRL, W[IS].

SG8 **Do not use spaces in front of `[]`, `()`, and to either side of `.` and `->`.**

Exception: there should be a space following keywords such if, for, while, switch, … , etc.

Example:

```
a->foo(); // Recommended
b.bar();  // Recommended

// Exception:
for ( i=1; i<n; i++ )
{
    // Do something
}
```

Source CERN.

4.2 Comments

SC2 **All comments should be written in complete (short and expressive) English sentences.**

The quality of the comments is an important factor in the understanding of the code.

Source CERN.

SC3 **In the header file, provide a comment describing the use of a declared function and attributes if this is not completely obvious from its name.**

Example:

```
class Point
{
    public:
    // Perpendicular distance of Point from Line
    Number distance (Line);
};
```

the comment includes the fact that it is the perpendicular distance.

Source CERN.

SC4 **In the implementation file, above each method implementation, provide a comment describing what the method does, how it does it (if not obvious), preconditions and postconditions.**

The code in a method will be much easier to understand and maintain if it is well explained in an initial comment.

Source CERN.

SC5 **All `#else` and `#endif` directives should carry a comment that tells what the corresponding `#if` was about if the conditional section is longer than five lines.**

The number five is obviously a reasonable arbitrary convention in order to make the item objective and checkable.

Example:

```
#ifndef GEOMETRY_POINT_H
#define GEOMETRY_POINT_H

class Point
{
    public:
        Point(Number x, Number y); // Create from (x,y)
        Number distance(Point point) const;
        Number distance(const Line & line) const;
        void translate(const Vector & vector);
};

#endif // GEOMETRY_POINT_H
```

Source CERN.

SC6 **Do not comment out old or obsolete code – remove it instead.**

A developer may do this temporarily during development but such code shall be removed before committing the code into the version control system. It is the job of the version control system to archive and track the old code, so attempting to archive the change history within the comments duplicates this function and does a poor job of it. Furthermore, such old code fragments tend to obfuscate the current version of the code and confuse the reader.

Source NRL.

SC7 **Each source file in a project shall contain a common-header comment block using doxygen markup. Other than filename, do not include information that can be easily extracted from the version control system such as author or modification date – this information is likely to quickly become outdated.**

Example:

```
//-------------------- Unclassified --------------------------

///
/// @file Example.h
///
/// Example class
///
/// Additional details about the Example class such as
/// references to specific requirements.

…
```

Source NRL.

5 C/C++ Compatibility

This section contains a set of conventions for interoperability between C and C++ modules.

LC1 **When creating a C module that is callable from C++ (or vice versa), disable name mangling in the module's header file when the file is included in C++.**

Example:

```
//-------------------- Unclassified --------------------------

///
/// @file Example.h
///

#ifdef __cplusplus
extern "C" {
#endif

void example_cFunction1( int i );
int  example_cFunction2( double x );

#ifdef __cplusplus
}
#endif

//-------------------- Unclassified --------------------------
```

Source NRL.

LC2 **Use a unique prefix to all public symbols in a C module (or C++ module callable from C).**

The C language does not provide namespaces to mitigate name collisions. Use a unique prefix (usually the module's name) to mitigate collisions. This also helps document which module owns the symbol.

Example:

```
int xyz_getNumberOfSamples();
```

Source NRL.

6 Example Files

The following simplified example files a, b, and c implement an “Xyz” class located in the “ns” namespace to illustrate some of the guidelines provided in this document. These files can serve as a basic template to start new C++ modules that follow this style.

Example a:
Header or specification file (Xyz.h):

```
/* -*- Mode: C; tab-width: 8; indent-tabs-mode: nil; c-basic-offset: 8 -*- */

/*-------------------------------UNCLASSIFIED-------------------------------*/

/**
 * @file
 * ns::Xyz class specification
 *
 */

#ifndef ns_Xyz_h
#define ns_Xyz_h

#include <string>


namespace ns
{

        /**
         * Brief description of ns::Xyz
         *
         * A longer description of ns::Xyz.
         *
         */
        class Xyz
        {

        public:
                Xyz();
                virtual ~Xyz();

                /*
                 * Property accessors
                 */
                double a() const;
                void setA( double a );
                double b() const;
                void setB( double b );
                std::string name() const;
                void setName( const std::string& name );

                /*
                 * Utility methods
                 */
                void rotate( double angleRads );

        private:
                double      mA;
                double      mB;
                std::string mName;

        };

}

/*
 * Inline methods and operators
 */
#include "Xyz.inl"


#endif /* ns_Xyz_h */

/*-------------------------------UNCLASSIFIED-------------------------------*/
```

Eaxample b:

Implementation file for inline methods (Xyz.inl):

```
/* -*- Mode: C; tab-width: 8; indent-tabs-mode: nil; c-basic-offset: 8 -*- */

/*-------------------------------UNCLASSIFIED-------------------------------*/

/**
 * @file
 * ns::Xyz inline methods
 *
 */


/*****************************************************************************/
/** Get value of "a".                                                        */
/*****************************************************************************/
inline double
ns::Xyz::a() const
{
        return mA;
}

/*****************************************************************************/
/** Set value of "a".                                                        */
/*****************************************************************************/
inline void
ns::Xyz::setA( double a )
{
        mA = a;
}

/*****************************************************************************/
/** Get value of "b".                                                        */
/*****************************************************************************/
inline double
ns::Xyz::b() const
{
        return mB;
}

/*****************************************************************************/
/** Set value of "b".                                                        */
/*****************************************************************************/
inline void
ns::Xyz::setB( double b )
{
        mB = b;
}

/*****************************************************************************/
/** Get value of "name".                                                     */
/*****************************************************************************/
inline std::string
ns::Xyz::name() const
{
        return mName;
}

/*****************************************************************************/
/** Set value of "name".                                                     */
/*****************************************************************************/
inline void
ns::Xyz::setName( const std::string& name )
{
        mName = name;
}


/*-------------------------------UNCLASSIFIED-------------------------------*/
```

Example c:

Implementation file (Xyz.cpp):

```
/* -*- Mode: C; tab-width: 8; indent-tabs-mode: nil; c-basic-offset: 8 -*- */

/*-------------------------------UNCLASSIFIED-------------------------------*/

/**
 * @file
 * ns::Xyz implementation
 *
 */

#include "Xyz.h"

#include <cmath>
#include <iostream>


/*****************************************************************************/
/** Constructor.                                                             */
/*****************************************************************************/
ns::Xyz::Xyz() : mA(0), mB(0), mName("Untitled")
{
        // Empty
}


/*****************************************************************************/
/** Destructor.                                                              */
/*****************************************************************************/
ns::Xyz::~Xyz()
{
        // Empty
}


/*****************************************************************************/
/** Rotate.                                                                  */
/*****************************************************************************/
void
ns::Xyz::rotate( double angleRads )
{
        double newA =  mA*cos( angleRads ) + mB*sin( angleRads );
        double newB = -mA*sin( angleRads ) + mB*cos( angleRads );

        mA = newA;
        mB = newB;
}


/*-------------------------------UNCLASSIFIED-------------------------------*/
```

7 Summary

The NRL Radar Division C++ Coding Standard was defined to be adhered to when writing C++ code. It is primarily addressed to all those involved in the production of C++ code for Naval Research Laboratory (NRL) Radar Division software projects such as the Office of Naval Research (ONR) Integrated Topside (InTop) Innovative Naval Prototype (INP) Flexible Distributed Array Radar (FlexDAR) backend software. It was originally directly derived from the CERN C++ Coding Standard Specification, Version 1.1, dated 5 January 2000 but has been significantly modified and updated. It also draws on other sources and includes many local modifications for NRL Radar Division applications.

The ISO 9000 and the Capability Maturity Model (CMM) state that coding standards are mandatory for any organization with quality goals. The purpose of this coding standard is to provide tools aimed at helping C++ programmers develop programs that are free of common types of errors, maintainable by different programmers, portable to other operating systems, easy to read and understand, and have a consistent style.

Questions of design, such as how to design a class or a class hierarchy, are beyond the scope of this coding standard. It is assumed here that the code is to be handwritten and not generated; otherwise a different coding standard would be needed for the input to the code generator. This document is not intended in any way as a substitute for the study of a book on C++ programming.